②

AD-A221 009 ~~ATION PAGE~~ | READ INSTRUCTIONS BEFORE COMPLETING FORM

| | 2. GOVT ACCESSION NO. | 3. RECIPIENT'S CATALOG NUMBER |

4. TITLE (and Subtitle)	5. TYPE OF REPORT & PERIOD COVERED
Ada Compiler Validation Summary Report: Meridian Software Systems, Inc., AdaVantage, Version 4.0, Apple Macintosh SE 30 (Host & Target), 900130W1.10238	30 Jan. 1990 to 30 Jan 1991
	6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s)	8. CONTRACT OR GRANT NUMBER(s)
Wright-Patterson AFB Dayton, OH, USA	

9. PERFORMING ORGANIZATION AND ADDRESS	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
Wright-Patterson AFB Dayton, OH, USA	

11. CONTROLLING OFFICE NAME AND ADDRESS	12. REPORT DATE
Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081	
	13. NUMBER OF PAGES

14. MONITORING AGENCY NAME & ADDRESS(If different from Controlling Office)	15. SECURITY CLASS (of this report) UNCLASSIFIED
Wright-Patterson AFB Dayton, OH, USA	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

UNCLASSIFIED

DTIC
ELECTE
APR 26 1990
S D

18. SUPPLEMENTARY NOTES

19. KEYWORDS (Continue on reverse side if necessary and identify by block number)

Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Meridian Software Systems, Inc., AdaVantage, Version 4.0, Wright-Patterson AFB, OH, Apple Machintosh SE 30 under System 6.0.3 (Host & Target), ACVC 1.10.
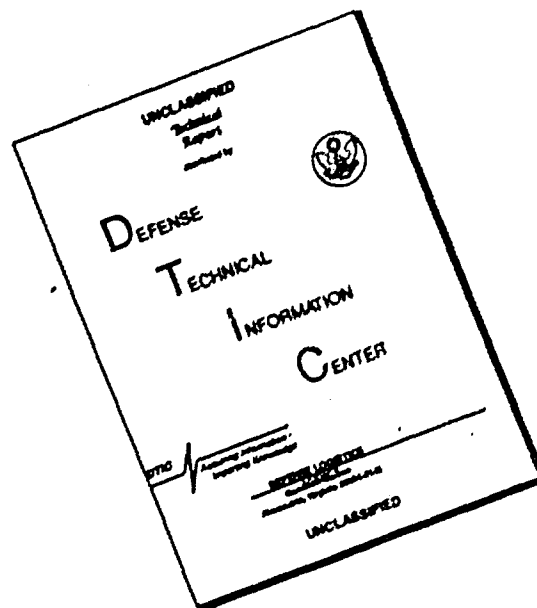
DD FORM 1473 EDITION OF 1 NOV 65 IS OBSOLETE
1 JAN 73 S/N 0102-LF-014-8601

90 04 24 090

DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 900130W1.10238
Meridian Software Systems, Inc.
AdaVantage, Version 4.0
Apple Macintosh SE 30

Completion of On-Site Testing:
30 January 1990

Prepared By:
Ada Validation Facility
ASD/SCOL
Wright-Patterson AFB OH 45433-6503

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington DC 20301-3081

Accesion For		
NTIS CRA&I		☑
DTIC TAB		☐
Unannounced		☐
Justification		
By		
Distribution /		
Availability Codes		
Dist	Avail and / or Special	
A-1		

Ada Compiler Validation Summary Report:

Compiler Name: AdaVantage, Version 4.0

Certificate Number: 900130W1.10238
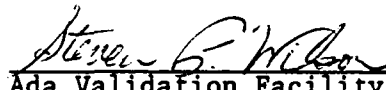
Host: Apple Macintosh SE 30 under
         System 6.0.3

Target: Apple Macintosh SE 30 under
         System 6.0.3

Testing Completed 30 January 1990 Using ACVC 1.10

Customer Agreement Number: 89-12-15-MSS

This report has been reviewed and is approved.

Steven G. Wilson

Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCOL
Wright-Patterson AFB OH   45433-6503

Jorge Narzullena

Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA   22311

Edward J. Lalliart

Ada Joint Program Office
Dr. John Solomond
Director
Department of Defense
Washington DC 20301

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation-dependent but is permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1-1

INTRODUCTION

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard

- To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard

- To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by SofTech, Inc. under the direction of the AVF according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO). On-site testing was completed 30 January 1990 at Irvine CA.

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C.#552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

> Ada Information Clearinghouse
> Ada Joint Program Office
> OUSDRE
> The Pentagon, Rm 3D-139 (Fern Street)
> Washington DC 20301-3081

or from:

> Ada Validation Facility
> ASD/SCOL
> Wright-Patterson AFB OH 45433-6503

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

> Ada Validation Organization
> Institute for Defense Analyses
> 1801 North Beauregard Street
> Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

2. Ada Compiler Validation Procedures, Version 2.0, Ada Joint Program Office, May 1989.

3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.

4. Ada Compiler Validation Capability User's Guide, December 1986.

1.4 DEFINITION OF TERMS

ACVC   The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.

Ada   An Ada Commentary contains all information relevant to the
Commentary   point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.

Ada Standard ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

Applicant   The agency requesting validation.

AVF   The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the Ada Compiler Validation Procedures.

AVO   The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.

Compiler   A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.

Failed test   An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.

Host   The computer on which the compiler resides.

Inapplicable   An ACVC test that uses features of the language that a
test   compiler is not required to support or may legitimately support in a way other than the one expected by the test.

Passed test   An ACVC test for which a compiler generates the expected result.

Target   The computer for which a compiler generates code.

Test   A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.

Withdrawn   An ACVC test found to be incorrect and not used to check
test   conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce compilation or link errors because of the way in which a program library is used at link time.

Class A tests ensure the successful compilation of legal Ada programs with certain language constructs which cannot be verified at compile time. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check the run time system to ensure that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters—for example, the number of identifiers permitted in a compilation or the number of units in a library—a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Class E tests are expected to execute successfully and check implementation-dependent options and resolutions of ambiguities in the Ada Standard. Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time—that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated. In some cases, an implementation may legitimately detect errors during compilation of the test.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of each test in the ACVC follows conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be

customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: AdaVantage, Version 4.0

ACVC Version: 1.10

Certificate Number: 900130W1.10238

Host Computer:

Machine:	Apple Macintosh SE 30
Operating System:	System 6.0.3
Memory Size:	2 Megabytes

Target Computer:

Machine:	Apple Macintosh SE 30
Operating System:	System 6.0.3
Memory Size:	2 Megabytes

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of
a compiler in those areas of the Ada Standard that permit implementations
to differ. Class D and E tests specifically check for such implementation
differences. However, tests in other classes also characterize an
implementation. The tests demonstrate the following characteristics:

   a. Capacities.

      (1) The compiler correctly processes a compilation containing 723
          variables in the same declarative part. (See test D29002K.)

      (2) The compiler correctly processes tests containing loop
          statements nested to 65 levels. (See tests D55A03A..H (8
          tests).)

      (3) The compiler correctly processes tests containing block
          statements nested to 65 levels. (See test D56001B.)

      (4) The compiler correctly processes tests containing recursive
          procedures separately compiled as subunits nested to 10
          levels. (See tests D64005E..G (3 tests).)

   b. Predefined types.

      (1) This implementation supports the additional predefined types
          SHORT_INTEGER and BYTE_INTEGER in package STANDARD. (See
          tests B86001T..Z (7 tests).)

   c. Expression evaluation.

      The order in which expressions are evaluated and the time at which
      constraints are checked are not defined by the language. While
      the ACVC tests do not specifically attempt to determine the order
      of evaluation of expressions, test results indicate the following:

      (1) None of the default initialization expressions for record
          components are evaluated before any value is checked for
          membership in a component's subtype. (See test C32117A.)

      (2) Assignments for subtypes are performed with the same precision
          as the base type. (See test C35712B.)

      (3) This implementation uses no extra bits for extra precision and
          uses all extra bits for extra range. (See test C35903A.)

(4) NUMERIC_ERROR is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

(5) No exception is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

(6) Underflow is gradual. (See tests C45524A..Z (26 tests).)

d. Rounding.

The method by which values are rounded in type conversions is not defined by the language. While the ACVC tests do not specifically attempt to determine the method of rounding, the test results indicate the following:

(1) The method used for rounding to integer is round to even. (See tests C46012A..Z (26 tests).)

(2) The method used for rounding to longest integer is round to even. (See tests C46012A..Z (26 tests).)

(3) The method used for rounding to integer in static universal real expressions is round away from zero. (See test C4A014A.)

e. Array types.

An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT.

For this implementation:

(1) Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises no exception. (See test C36003A.)

(2) NUMERIC_ERROR is raised when 'LENGTH is applied to an array type with INTEGER'LAST + 2 components with each component being a null array. (See test C36202A.)

(3) NUMERIC_ERROR is raised when 'LENGTH is applied to an array type with SYSTEM.MAX_INT + 2 components with each component being a null array. (See test C36202B.)

(4) A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises NUMERIC_ERROR when the array objects are declared. (See test C52103X.)

(5) A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises NUMERIC_ERROR when the array objects are declared. (See test C52104Y.)

(6) A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises NUMERIC_ERROR when the array type is declared. (See test E52103Y.)

(7) In assigning one-dimensional array types, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

(8) In assigning two-dimensional array types, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

f. Discriminated types.

(1) In assigning record types with discriminants, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

g. Aggregates.

(1) In the evaluation of a multi-dimensional aggregate, all choices are evaluated before checking against the index type. (See tests C43207A and C43207B.)

(2) In the evaluation of an aggregate containing subaggregates, not all choices are evaluated before being checked for identical bounds. (See test E43212B.)

(3) CONSTRAINT_ERROR is raised before all choices are evaluated when a bound in a non-null range of a non-null aggregate does not belong to an index subtype. (See test E43211B.)

h. Pragmas.

(1) The pragma INLINE is not supported for functions or procedures. (See tests LA3004A..B (2 tests), EA3004C..D (2 tests), and CA3004E..F (2 tests).)

i. Generics.

    (1) Generic unit declarations, bodies, and subunits can be compiled in separate compilations. (See tests CA1012A and CA3011A.)

    (2) If a generic unit body or one of its subunits is compiled or recompiled after the generic unit is instantiated, the unit instantiating the generic is made obsolete. The obsolesence is recognized at binding time, and the binding is stopped. (See tests CA2009C, CA2009F, BC3204C, and BC3205D.)

j. Input and output.

    (1) The package SEQUENTIAL_IO cannot be instantiated with unconstrained array types or record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)

    (2) The package DIRECT_IO cannot be instantiated with unconstrained array types or record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)

    (3) Modes IN_FILE and OUT_FILE are supported for SEQUENTIAL_IO. (See tests CE2102D..E (2 tests), CE2102N, and CE2102P.)

    (4) Modes IN_FILE, OUT_FILE, and INOUT_FILE are supported for DIRECT_IO. (See tests CE2102F, CE2102I..J (2 tests), CE2102R, CE2102T, and CE2102V.)

    (5) Modes IN_FILE and OUT_FILE are supported for text files. (See tests CE3102E and CE3102I..K (3 tests).)

    (6) RESET and DELETE operations are supported for SEQUENTIAL_IO. (See tests CE2102G and CE2102X.)

    (7) RESET and DELETE operations are supported for DIRECT_IO. (See tests CE2102K and CE2102Y.)

    (8) RESET and DELETE operations are supported for text files. (See tests CE3102F..G (2 tests), CE3104C, CE3110A, and CE3114A.)

    (9) Overwriting to a sequential file does not truncate the file. (See test CE2208B.)

   (10) Temporary sequential files are given names and deleted when closed. (See test CE2108A.)

   (11) Temporary direct files are given names and deleted when closed. (See test CE2108C.)

CONFIGURATION INFORMATION

(12) Temporary text files are given names and deleted when closed.
     (See test CE3112A.)

(13) More than one internal file can be associated with each
     external file for sequential files when reading only. (See
     tests CE2107A..E (5 tests), CE2102L, CE2110B, and CE2111D.)

(14) More than one internal file can be associated with each
     external file for direct files when reading only. (See tests
     CE2107F..H (3 tests), CE2110D, and CE2111H.)

(15) More than one internal file can be associated with each
     external file for text files when reading only. (See tests
     CE3111A..E (5 tests), CE3114B, and CE3115A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.10 of the ACVC comprises 3717 tests. When this compiler was tested, 44 tests had been withdrawn because of test errors. The AVF determined that 286 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation. Modifications to the code, processing, or grading for 4 tests were required to successfully demonstrate the test objective.

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	A	B	TEST CLASS C	D	E	L	TOTAL
Passed	127	1129	2049	16	22	44	3387
Inapplicable	2	9	266	1	6	2	286
Withdrawn	1	2	35	0	6	0	44
TOTAL	130	1140	2350	17	34	46	3717

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT						CHAPTER									TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14		
Passed	198	576	545	245	171	99	160	332	129	36	250	368	278	3387
Inappl	14	73	135	3	1	0	6	0	8	0	2	1	43	286
Wdrn	1	1	0	0	0	0	0	2	0	0	1	35	4	44
TOTAL	213	650	680	248	172	99	166	334	137	36	253	404	325	3717

3.4 WITHDRAWN TESTS

The following 44 tests were withdrawn from ACVC Version 1.10 at the time of this validation:

E28005C	A39005G	B97102E	C97116A	BC3009B	CD2A62D
CD2A63A	CD2A63B	CD2A63C	CD2A63D	CD2A66A	CD2A66B
CD2A66C	CD2A66D	CD2A73A	CD2A73B	CD2A73C	CD2A73D
CD2A76A	CD2A76B	CD2A76C	CD2A76D	CD2A81G	CD2A83G
CD2A84M	CD2A84N	CD2B15C	CD2D11B	CD5007B	CD50110
ED7004B	ED7005C	ED7005D	ED7006C	ED7006D	CD7105A
CD7203B	CD7204B	CD7205C	CD7205D	CE2107I	CE3111C
CE3301A	CE3411B				

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 286 tests were inapplicable for the reasons indicated:

   a. The following 201 tests are not applicable because they have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

       C24113L..Y (14 tests)   C35705L..Y (14 tests)
       C35706L..Y (14 tests)   C35707L..Y (14 tests)
       C35708L..Y (14 tests)   C35802L..Z (15 tests)

C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

b. C35702A and B86001T are not applicable because this implementation supports no predefined type SHORT_FLOAT.

c. C35702B and B86001U are not applicable because this implementation supports no predefined type LONG_FLOAT.

d. The following 16 tests are not applicable because this implementation does not support a predefined type LONG_INTEGER:

C45231C	C45304C	C45502C	C45503C	C45504C
C45504F	C45611C	C45613C	C45614C	C45631C
C45632C	B52004D	C55B07A	B55B09C	B86001W
CD7101F				

e. C45531M..P (4 tests) and C45532M..P (4 tests) are not applicable because the value of SYSTEM.MAX_MANTISSA is less than 48.

f. D64005G is not applicable because this implementation does not support nesting 17 levels of recursive procedure calls.

g. C86001F is not applicable because, for this implementation, the package TEXT_IO is dependent upon package SYSTEM. This test recompiles package SYSTEM, making package TEXT_IO, and hence package REPORT, obsolete.

h. B86001Y is not applicable because this implementation supports no predefined fixed-point type other than DURATION.

i. B86001Z is not applicable because this implementation supports no predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

j. CA2009C, CA2009F, BC3204C, and BC3205D are not applicable because this implementation does not support separate compilation of generic specifications, bodies, or subunits, if an instantiation is given before compilation of its bodies or subunits. The created dependency is detected at bind time.

k. LA3004A, LA3004B, EA3004C, EA3004D, CA3004E, and CA3004F are not applicable because this implementation does not support pragma INLINE.

l. AE2101C and EE2201D..E (2 tests) use instantiations of package SEQUENTIAL_IO with unconstrained array types and record types with discriminants without defaults. These instantiations are rejected by this compiler.

m. AE2101H, EE2401D, and EE2401G use instantiations of package DIRECT_IO with unconstrained array types and record types with discriminants without defaults. These instantiations are rejected by this compiler.

n. CE2102D is inapplicable because this implementation supports CREATE with IN_FILE mode for SEQUENTIAL_IO.

o. CE2102E is inapplicable because this implementation supports CREATE with OUT_FILE mode for SEQUENTIAL_IO.

p. CE2102F is inapplicable because this implementation supports CREATE with INOUT_FILE mode for DIRECT_IO.

q. CE2102I is inapplicable because this implementation supports CREATE with IN_FILE mode for DIRECT_IO.

r. CE2102J is inapplicable because this implementation supports CREATE with OUT_FILE mode for DIRECT_IO.

s. CE2102N is inapplicable because this implementation supports OPEN with IN_FILE mode for SEQUENTIAL_IO.

t. CE2102O is inapplicable because this implementation supports RESET with IN_FILE mode for SEQUENTIAL_IO.

u. CE2102P is inapplicable because this implementation supports OPEN with OUT_FILE mode for SEQUENTIAL_IO.

v. CE2102Q is inapplicable because this implementation supports RESET with OUT_FILE mode for SEQUENTIAL_IO.

w. CE2102R is inapplicable because this implementation supports OPEN with INOUT_FILE mode for DIRECT_IO.

x. CE2102S is inapplicable because this implementation supports RESET with INOUT_FILE mode for DIRECT_IO.

y. CE2102T is inapplicable because this implementation supports OPEN with IN_FILE mode for DIRECT_IO.

z. CE2102U is inapplicable because this implementation supports RESET with IN_FILE mode for DIRECT_IO.

aa. CE2102V is inapplicable because this implementation supports OPEN with OUT_FILE mode for DIRECT_IO.

ab. CE2102W is inapplicable because this implementation supports RESET with OUT_FILE mode for DIRECT_IO.

ac. CE3102E is inapplicable because this implementation supports CREATE with IN_FILE mode for text files.

ad. CE3102F is inapplicable because this implementation supports RESET for text files.

ae. CE3102G is inapplicable because this implementation supports deletion of an external file for text files.

af. CE3102I is inapplicable because this implementation supports CREATE with OUT_FILE mode for text files.

ag. CE3102J is inapplicable because this implementation supports OPEN with IN_FILE mode for text files.

ah. CE3102K is inapplicable because this implementation supports OPEN with OUT_FILE mode for text files.

ai. CE2107B..E (4 tests), CE2107L, and CE2110B are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for sequential files. The proper exception is raised when multiple access is attempted.

aj. CE2107G..H (2 tests), CE2110D, CE2111D, and CE2111H are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for direct files. The proper exception is raised when multiple access is attempted.

ak. CE3111B, CE3111D..E (2 tests), CE3114B, and CE3115A are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for text files. The proper exception is raised when multiple access is attempted.

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that wasn't anticipated by the test (such as raising one exception instead of another).

Modifications were required for 4 tests.

The following tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

    B22003A   B49003A   B49005A   B85013D

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.10 produced by the AdaVantage, Version 4.0 compiler was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the AdaVantage, Version 4.0 compiler using ACVC Version 1.10 was conducted on-site by a validation team from the AVF. The configuration in which the testing was performed is described by the following designations of hardware and software components:

Host computer:	Apple Macintosh SE 30
Host operating system:	System 6.0.3
Target computer:	Apple Macintosh SE 30
Target operating system:	System 6.0.3
Compiler:	AdaVantage, Version 4.0

A set of diskettes containing all tests except for withdrawn tests and tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the diskettes. Tests requiring modifications during the prevalidation testing were included in their modified form on the diskettes.

The contents of the diskettes were loaded directly onto the host computer.

After the test files were loaded to disk, the full set of tests was compiled, linked, and all executable tests were run on the Apple Macintosh SE 30. Results were transferred to a Sun 3 computer via a serial line and the KERMIT file transfer program, and then printed.

The compiler was tested using command scripts provided by Meridian Software Systems, Inc. and reviewed by the validation team. The compiler was tested using all the following option settings. See Appendix E for a complete listing of the compiler options for this implementation. The following list of compiler options includes those options which were invoked explicitly:

ada -fE	Generate error file for the Ada listing utility (alu).
ada -fI	Ignore compilation errors and continue to generate code for compilation units that have no errors (for test EA1003B).
ada -fQ	Suppress "added to library" and "Generating code for" information messages.
ada -fw	Suppress informative warning messages.
alu -c	Produce continuous form Ada listings (no page headers).
alu -p	Obey PRAGMA PAGE directives within program even though the -c flag says not to generate page breaks.
alu -s	Output Ada listing to the standard output file instead of to a disk file.

Tests were compiled, linked, and executed (as appropriate) using 2 computers. Test output, compilation listings, and job logs were captured on diskettes and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

Testing was conducted at Irvine CA and was completed on 30 January 1990.

APPENDIX A

DECLARATION OF CONFORMANCE

Meridian Software Systems, Inc. has submitted the
following Declaration of Conformance concerning the
AdaVantage, Version 4.0 compiler.
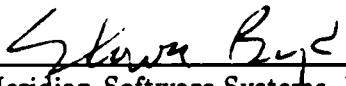
DECLARATION OF CONFORMANCE

Compiler Implementor: Meridian Software Systems, Inc.
Ada Validation Facility: ASD/SCEL, Wright-Patterson AFB OH 45433-6503
Ada Compiler Validation Capability (ACVC) Version: 1.10

Base Configuration

Base Compiler Name:	AdaVantage
Version:	4.0
Host Architecture ISA:	Apple Macintosh SE 30
OS&VER #:	System 6.0.3
Target Architecture ISA:	Apple Macintosh SE 30
OS&VER #:	System 6.0.3

Implementor's Declaration

I, the undersigned, representing Meridian Software Systems, Inc., have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compiler(s) listed in this declaration. I declare that Meridian Software Systems, Inc. is the owner of record of the Ada language compiler(s) listed above and, as such, is responsible for maintaining said compiler(s) in conformance to ANSI/MIL-STD-1815A. All certificates and registrations for Ada language compiler(s) listed in this declaration shall be made only in the owner's corporate name.
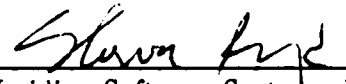
Date: 19 Dec 89

Meridian Software Systems, Inc.
Stowe Boyd, Vice President of Research and Development

Owner's Declaration

I, the undersigned, representing Meridian Software Systems, Inc., take full responsibility for implementation and maintenance of the Ada compiler(s) listed above, and agree to the public disclosure of the final Validation Summary Report. I declare that all of the Ada language compilers listed, and their host/target performance are in compilance with the Ada Language Standard ANSI/MIL-STD-1815A.

Date: 19 Dec 89

Meridian Software Systems, Inc.
Stowe Boyd, Vice President of Research and Development

A-2

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to
implementation-dependent pragmas, to certain machine-dependent conventions
as mentioned in Chapter 13 of the Ada Standard, and to certain allowed
restrictions on representation clauses. The implementation-dependent
characteristics of the AdaVantage, Version 4.0 compiler, as described in
this Appendix, are provided by Meridian Software Systems, Inc. Unless
specifically noted otherwise, references in this Appendix are to compiler
documentation and not to this report. Implementation-specific portions of
the package STANDARD, which are not a part of Appendix F, are:

package STANDARD is

    ...

    type INTEGER is range -2_147_483_648 .. 2_147_483_647;
    type SHORT_INTEGER is range -32768 .. 32767;
    type BYTE_INTEGER is range -128 .. 127;

    type FLOAT is digits 15
        range -1.79769313486231E+308 .. 1.79769313486231E+308;

    type DURATION is delta 0.0001 range -86400.0 .. 86400.0;

    ...

end STANDARD;

Implementation-Dependent Characteristics

This appendix lists implementation-dependent characteristics of the Macintosh version of Meridian AdaVantage. Note that there are no preceding appendices. This appendix is called "Appendix F" because of a very clearly stated requirement by ANSI/MIL-STD-1815A that this appendix be so named.

Implemented Chapter 13 features include length clauses, enumeration representation clauses, record representation clauses, address clauses, interrupts, package **SYSTEM**, machine code insertions, pragma **interface**, and unchecked programming.

F.1   Pragmas

The implemented pre-defined pragmas are:

elaborate	Implemented as per ANSI/MIL-STD-1815A section 10.5.
interface	See section F.1.1.
list	Implemented as per ANSI/MIL-STD-1815A Appendix B.
pack	See section F.1.2.
page	Implemented as per ANSI/MIL-STD-1815A Appendix B.
priority	Implemented as per ANSI/MIL-STD-1815A Appendix B.
suppress	See section F.1.3.

The remaining pre-defined pragmas are accepted, but presently ignored:

controlled	optimize	system_name
inline	shared	
memory_size	storage_unit	

Named parameter notation for pragmas is not supported.

When illegal parameter forms are encountered at compile time, the compiler issues a warning message rather than an error, as required by the Ada language definition.

Refer to ANSI/MIL-STD-1815A Appendix B for additional information about the pre-defined pragmas.

F.1.1   Pragma Interface

The form of pragma interface in Meridian AdaVantage is:

```
pragma interface( language, subprogram [, "link-name"] );
```

where:

      language   is the interface language, one of the names assembly, c, builtin, or internal. The names builtin and

internal are reserved for use by Meridian compiler maintainers in run-time support packages.

subprogram is the name of a subprogram to which the pragma **interface** applies. If *link-name* is omitted, then the Ada subprogram name is also used as the object code symbol name. Depending on the *language* specified, some automatic modifications may be made to the object code symbol name.

link-name is an optional string literal specifying the name of the non-Ada subprogram corresponding to the Ada subprogram named in the second parameter. The *link-name* is used as the object code symbol. Depending on the *language* specified, some automatic modifications may be made to the object code symbol name.

It is appropriate to use the optional *link-name* parameter to pragma **interface** when the interface subprogram has a name that does not correspond at all to its Ada identifier or when the interface subprogram name cannot be given using rules for constructing Ada identifiers (e.g. if the name contains a '$' character).

The characteristics of object code symbols emitted for each interface language are:

assembly The object code symbol is the same as the subprogram name. If no *link-name* string is specified, then the subprogram name is translated to lower case.

builtin The object code symbol is the same as the subprogram name, but with one underscore character ('_') prepended, whether or not a *link-name* string is specified. If no *link-name* string is specified, then the subprogram name is translated to lower case. This language interface is reserved for special interfaces defined

by Meridian Software Systems, Inc. The `builtin` interface is presently used to declare certain low-level runtime operations whose names must not conflict with programmer-defined or system-defined names.

c   The object code symbol is the same as the subprogram name. If no *link-name* string is specified, then the subprogram name is translated to lower case. This is the convention used by the Macintosh C compiler.

internal   No object code symbol is emitted for an internal language interface; this language interface is reserved for special interfaces defined by Meridian Software Systems, Inc. The `internal` interface is presently used to declare certain machine-level bit operations.

In no case are the low-level calling conventions changed; no automatic data conversions are performed on parameters of interface subprograms. It is up to the programmer to ensure that calling conventions match and that any necessary data conversions take place when calling interface subprograms.

A pragma `interface` may appear within the same declarative part as the subprogram to which the pragma `interface` applies, following the subprogram declaration, and prior to the first use of the subprogram. A pragma `interface` that applies to a subprogram declared in a package specification must occur within the same package specification as the subprogram declaration; the pragma `interface` may not appear in the package body in this case. A pragma `interface` declaration for either a private or non-private subprogram declaration may appear in the private part of a package specification.

Pragma `interface` for library units is not supported.

Refer to ANSI/MIL-STD-1815A section 13.9 for additional information about pragma `interface`.

F.1.2 Pragma Pack

Pragma pack is implemented for composite types (records and arrays).

Pragma pack is permitted following the composite type declaration to which it applies, provided that the pragma occurs within the same declarative part as the composite type declaration, before any objects or components of the composite type are declared.

Note that the declarative part restriction means that the type declaration and accompanying pragma pack cannot be split across a package specification and body.

The effect of pragma pack is to minimize storage consumption by discrete component types whose ranges permit packing. Use of pragma pack does not defeat allocations of alignment storage gaps for some record types. Pragma pack does not affect the representations of real types, pre-defined integer types, and access types.

F.1.3 Pragma Suppress

Pragma suppress is implemented as described in ANSI/MIL-STD-1815A section 11.7, with these differences:

- Presently, **division_check** and **overflow_check** must be suppressed via a compiler flag, **-fN**; pragma **suppress** is ignored for these two numeric checks.

- The optional "**ON =>**" parameter name notation for pragma **suppress** is ignored.

- The optional second parameter to pragma **suppress** is ignored; the pragma always applies to the entire scope in which it appears.

F.2   Attributes

There are presently no implementation-dependent attributes in Meridian
AdaVantage. All attributes described in ANSI/MIL-STD-1815A Appendix
A are implemented.

F.3   Standard Types

Two additional standard types are defined in AdaVantage:

1. SHORT_INTEGER, defined with less precision than type INTEGER.

2. BYTE_INTEGER, defined with less precision than type SHORT_INTEGER;

The standard numeric types are defined as:

```
type BYTE_INTEGER is range -128..127;

type SHORT_INTEGER is range -32768 .. 32767;

type INTEGER is range -2147483648 .. 2147483647;

type FLOAT is digits 15
    range -1.79769313486231E+308 .. 1.79769313486231E+308;

type DURATION is delta 0.0001
    range -86400.0000 .. 86400.0000;
```


F.4   Package SYSTEM

The specification of package SYSTEM for Macintosh is:

```
package SYSTEM is
```

```
type ADDRESS is new INTEGER;

type NAME is ( m68000 );
SYSTEM_NAME   : constant NAME := m68000;

STORAGE_UNIT  : constant := 8;
MEMORY_SIZE   : constant := 1024;

-- System-Dependent Named Numbers

MIN_INT       : constant := -2147483648;
MAX_INT       : constant := 2147483647;
MAX_DIGITS    : constant := 15;
MAX_MANTISSA  : constant := 31;
FINE_DELTA    : constant := 2.0 ** (-31);
TICK          : constant := 1.0;

-- Other System-Dependent Declarations

subtype PRIORITY is INTEGER range 1 .. 20;

end SYSTEM;
```

The value of SYSTEM.MEMORY_SIZE is presently meaningless.

F.5   Restrictions on Representation Clauses

F.5.1   Length Clauses

A size specification (T'SIZE) is rejected if fewer bits are specified than can accommodate the type. The minimum size of a composite type may be subject to application of pragma pack. It is permitted to specify precise sizes for unsigned integer ranges, e.g. 8 for the range 0..255. However, because of requirements imposed by the Ada language definition, a full 32-bit range of unsigned values, i.e. 0..(2**32)-1, cannot be defined, even using a size specification.

The specification of collection size (T'STORAGE_SIZE) is evaluated at runtime when the scope of the type to which the length clause applies is entered, and is therefore subject to rejection (via STORAGE_ERROR) based on available

storage at the time the allocation is made. A collection may include storage used for run-time administration of the collection, and therefore should not be expected to accommodate a specific number of objects. Furthermore, certain classes of objects such as unconstrained discriminant array components of records may be allocated outside a given collection, so a collection may accommodate more objects than might be expected.

The specification of storage for a task activation (T'STORAGE_SIZE) is evaluated at run-time when a task to which the length clause applies is activated, and is therefore subject to rejection (via STORAGE_ERROR) based on available storage at the time the allocation is made. Storage reserved for a task activation is separate from storage needed for any collections defined within a task body.

The specification of *small* for a fixed point type (T'SMALL) is subject only to restrictions defined in ANSI/MIL-STD-1815A section 13.2.

F.5.2   Enumeration Representation Clauses

The internal code for the literal of an enumeration type named in an enumeration representation clause must be in the range of STANDARD.INTEGER.

The value of an internal code may be obtained by applying an appropriate instantiation of UNCHECKED_CONVERSION to an integer type.

F.5.3   Record Representation Clauses

The storage unit offset (the at *static_simple_expression* part) is given in terms of 8-bit storage units and must be even.

A bit position (the range part) applied to a discrete type component may be in the range 0..15, with 0 being the least significant bit of a component. A range specification may not specify a size smaller than can accommodate the component. A range specification for a component not accommodating bit packing may have a higher upper bound as appropriate (e.g. 0..31 for

a discriminant string component); the lower bound for such a type must be zero. Refer to the internal data representation of a given component in determining the component size and assigning offsets.

Components of discrete types for which bit positions are specified may not straddle 16-bit word boundaries.

The value of an alignment clause (the optional at mod part) must evaluate to 1, 2, 4, or 8, and may not be smaller than the highest alignment required by any component of the record. On Macintosh, this means that some records may not have alignment clauses smaller than 2.

F.5.4   Address Clauses

An address clause may be supplied for an object (whether constant or variable) or a task entry, but not for a subprogram, package, or task unit. The meaning of an address clause supplied for a task entry is given in section F.5.5

An address expression for an object is a 32-bit linear memory address of type SYSTEM.ADDRESS.

F.5.5   Interrupts

A task entry's address clause can be used to associate the entry with a signal a la UNIX. Values in the range 0..16 may be specified. At present, only signal 2, representing the Command-Dot keyboard interrupt (or SIGINT signal), can be activated via the signal mechanism. Value 0 may be specified, but no signal corresponds to value 0, and such an entry is never called via the signal mechanism.

An interrupt entry may not have any parameters.

F.5.6 Change of Representation

A change of representation effected by means of type conversion between objects of two types, one type derived from a parent type for which a representation clause has been supplied (as described in ANSI/MIL-STD-1815A section 13.6), is not permitted.

F.6 Implementation-Dependent Components

No names are generated by the implementation to denote implementation-dependent components.

F.7 Unchecked Conversions

There are no restrictions on the use of UNCHECKED_CONVERSION. Conversions between objects whose sizes do not conform may result in storage areas with undefined values.

F.8 Input-Output Packages

A summary of the implementation-dependent input-output characteristics is:

- In calls to OPEN and CREATE, the FORM parameter must be the empty string (the default value).

- More than one internal file can be associated with a single external file for reading only. For writing, only one internal file may be associated with an external file; RESET may not be used to get around this rule.

- Temporary sequential and direct files are given names. Temporary files are deleted when they are closed.

- File I/O is buffered; text files associated with terminal devices are line-buffered.

- The packages **SEQUENTIAL_IO** and **DIRECT_IO** cannot be instantiated with unconstrained composite types or record types with discriminants without defaults.

F.9   Source Line and Identifier Lengths

Source lines and identifiers in Ada source programs are presently limited to 200 characters in length.

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

Name and Meaning	Value
$ACC_SIZE An integer literal whose value is the number of bits sufficient to hold any value of an access type.	32
$BIG_ID1 An identifier the size of the maximum input line length which is identical to $BIG_ID2 except for the last character.	(1..199 => 'A', 200 => '1')
$BIG_ID2 An identifier the size of the maximum input line length which is identical to $BIG_ID1 except for the last character.	(1..199 => 'A', 200 => '2')
$BIG_ID3 An identifier the size of the maximum input line length which is identical to $BIG_ID4 except for a character near the middle.	(1..99 => 'A', 100 => '3', 101..200 => 'A')

TEST PARAMETERS

Name and Meaning	Value
$BIG_ID4 An identifier the size of the maximum input line length which is identical to $BIG_ID3 except for a character near the middle.	(1..99 => 'A', 100 => '4', 101..200 => 'A')
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	(1..197 => '0', 198..200 => "298")
$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	(1..195 => '0', 196..200 => "690.0")
$BIG_STRING1 A string literal which when catenated with $BIG_STRING2 yields the image of $BIG_ID1.	(1 => '"', 2..101 => 'A', 102 => '"')
$BIG_STRING2 A string literal which when catenated to the end of $BIG_STRING1 yields the image of $BIG_ID1.	(1 => '"', 2..100 => 'A', 101 => '1', 102 => '"')
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	(1..180 => ' ')
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2_147_483_646
$DEFAULT_MEM_SIZE An integer literal whose value is SYSTEM.MEMORY_SIZE.	1024
$DEFAULT_STOR_UNIT An integer literal whose value is SYSTEM.STORAGE_UNIT.	8

Name and Meaning	Value
$DEFAULT_SYS_NAME The value of the constant SYSTEM.SYSTEM_NAME.	M68000
$DELTA_DOC A real literal whose value is SYSTEM.FINE_DELTA.	2.0**(-31)
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	2_147_483_647
$FIXED_NAME The name of a predefined fixed-point type other than DURATION.	NO_SUCH_FIXED_TYPE
$FLOAT_NAME The name of a predefined floating-point type other than FLOAT, SHORT_FLOAT, or LONG_FLOAT.	NO_SUCH_FLOAT_TYPE
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	90_000.0
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	10_000_000.0
$HIGH_PRIORITY An integer literal whose value is the upper bound of the range for the subtype SYSTEM.PRIORITY.	20
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	NODIRECTORY:FILENAME1
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	NODIRECTORY:FILENAME2
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-2147483648

TEST PARAMETERS

Name and Meaning	Value
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	2147483647
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST + 1.	2_147_483_648
$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-90_000.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-10_000_000.0
$LOW_PRIORITY An integer literal whose value is the lower bound of the range for the subtype SYSTEM.PRIORITY.	1
$MANTISSA_DOC An integer literal whose value is SYSTEM.MAX_MANTISSA.	31
$MAX_DIGITS Maximum digits supported for floating-point types.	15
$MAX_IN_LEN Maximum input line length permitted by the implementation.	200
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2_147_483_648
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be $MAX_IN_LEN long.	(1..2 => "2:", 3..197 => '0', 198..200 => "11:")

Name and Meaning	Value
$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be $MAX_IN_LEN long.	(1..3 => "16:", 4..196 => '0', 197..200 => "F.E:")
$MAX_STRING_LITERAL A string literal of size $MAX_IN_LEN, including the quote characters.	(1 => '"', 2..199 => 'A', 200 => '"')
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-2147483648
$MIN_TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has no entries, no declarations, and "NULL;" as the only statement in its body.	32
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	BYTE_INTEGER
$NAME_LIST A list of enumeration literals in the type SYSTEM.NAME, separated by commas.	M68000
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFE#
$NEW_MEM_SIZE An integer literal whose value is a permitted argument for pragma MEMORY_SIZE, other than $DEFAULT_MEM_SIZE. If there is no other value, then use $DEFAULT_MEM_SIZE.	1024

C-5

TEST PARAMETERS

Name and Meaning	Value
$NEW_STOR_UNIT An integer literal whose value is a permitted argument for pragma STORAGE_UNIT, other than $DEFAULT_STOR_UNIT. If there is no other permitted value, then use value of SYSTEM.STORAGE_UNIT.	8
$NEW_SYS_NAME A value of the type SYSTEM.NAME, other than $DEFAULT_SYS_NAME. If there is only one value of that type, then use that value.	M68000
$TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has a single entry with one 'IN OUT' parameter.	32
$TICK A real literal whose value is SYSTEM.TICK.	1.0

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 44 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form AI-ddddd is to an Ada Commentary.

a. E28005C: This test expects that the string "-- TOP OF PAGE. --63" of line 204 will appear at the top of the listing page due to a pragma PAGE in line 203; but line 203 contains text that follows the pragma, and it is this text that must appear at the top of the page.

b. A39005G: This test unreasonably expects a component clause to pack an array component into a minimum size (line 30).

c. B97102E: This test contains an unintended illegality: a select statement contains a null statement at the place of a selective wait alternative (line 31).

d. C97116A: This test contains race conditions, and it assumes that guards are evaluated indivisibly. A conforming implementation may use interleaved execution in such a way that the evaluation of the guards at lines 50 & 54 and the execution of task CHANGING_OF_THE_GUARD results in a call to REPORT.FAILED at one of lines 52 or 56.

e. BC3009B: This test wrongly expects that circular instantiations will be detected in several compilation units even though none of the units is illegal with respect to the units it depends on; by AI-00256, the illegality need not be detected until execution is attempted (line 95).

f. CD2A62D: This test wrongly requires that an array object's size be no greater than 10 although its subtype's size was specified to be 40 (line 137).

g. CD2A63A..D, CD2A66A..D, CD2A73A..D, and CD2A76A..D (16 tests): These tests wrongly attempt to check the size of objects of a derived type (for which a 'SIZE length clause is given) by passing them to a derived subprogram (which implicitly converts them to the parent type (Ada standard 3.4:14)). Additionally, they use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

h. CD2A81G, CD2A83G, CD2A84M..N, and CD50110 (5 tests): These tests assume that dependent tasks will terminate while the main program executes a loop that simply tests for task termination; this is not the case, and the main program may loop indefinitely (lines 74, 85, 86, 96, and 58, respectively).

i. CD2B15C and CD7205C: These tests expect that a 'STORAGE_SIZE length clause provides precise control over the number of designated objects in a collection; the Ada standard 13.2:15 allows that such control must not be expected.

j. CD2D11B: This test gives a SMALL representation clause for a derived fixed-point type (at line 30) that defines a set of model numbers that are not necessarily represented in the parent type; by Commentary AI-00099, all model numbers of a derived fixed-point type must be representable values of the parent type.

k. CD5007B: This test wrongly expects an implicitly declared subprogram to be at the address that is specified for an unrelated subprogram (line 303).

l. ED7004B, ED7005C..D, and ED7006C..D (5 tests): These tests check various aspects of the use of the three SYSTEM pragmas; the AVO withdraws these tests as being inappropriate for validation.

m. CD7105A: This test requires that successive calls to CALENDAR.CLOCK change by at least SYSTEM.TICK; however, by Commentary AI-00201, it is only the expected frequency of change that must be at least SYSTEM.TICK—particular instances of change may be less (line 29).

n. CD7203B and CD7204B: These tests use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

o. CD7205D: This test checks an invalid test objective: it treats the specification of storage to be reserved for a task's activation as though it were like the specification of storage for a collection.

p. CE2107I: This test requires that objects of two similar scalar types be distinguished when read from a file--DATA_ERROR is expected to be raised by an attempt to read one object as of the other type. However, it is not clear exactly how the Ada standard 14.2.4:4 is to be interpreted; thus, this test objective is not considered valid (line 90).

q. CE3111C: This test requires certain behavior, when two files are associated with the same external file, that is not required by the Ada standard.

r. CE3301A: This test contains several calls to END_OF_LINE and END_OF_PAGE that have no parameter: these calls were intended to specify a file, not to refer to STANDARD_INPUT (lines 103, 107, 118, 132, and 136).

s. CE3411B: This test requires that a text file's column number be set to COUNT'LAST in order to check that LAYOUT_ERROR is raised by a subsequent PUT operation. But the former operation will generally raise an exception due to a lack of available disk space, and the test would thus encumber validation testing.

APPENDIX E

COMPILER OPTIONS AS SUPPLIED BY MERIDIAN

Compiler: AdaVantage, Version 4.0

ACVC Version: 1.10

- Functional Changes

  A new switch has been added to the ada command to control the calling
  sequences for "pragma interface C". The -mc68881 option to ada allows
  you to also compile your C functions with the same option. AdaVantage
  still maintains floating point objects in the 64-bit format, but now you
  can call C routines that use either the 80-bit format (default) or the 96-bit
  format.

14.1 ada

Name: **ada**—invoke the Meridian AdaVantage compiler

14.1.1 Invocation

ada [*option ...*] *file*.**ada** ...

14.1.2 Description

The **ada** command invokes the Meridian AdaVantage compiler.

A program library must be created using **mklib** or **newlib** in advance of any compilation. The compiler aborts if it is unable to find a program library (either the default, **ada.lib**, in the current working directory or the library name specified with the **-L** option).

Note that the source file has the extension **.ada**. Just about any non–empty file extension is permitted. The ones not allowed include those used by the AdaVantage compiling system for other purposes such as **.o** for object module files. If an illegal extension is given, the error message "illegal source file extension" is displayed. Some other commonly used source file extensions are:

    .ads   for package specification source files

    .adb   for package body source files

    .sub   for subunit (separate) source files

The Meridian AdaVantage compiler emits Assembly language code that is then automatically compiled by the MPW Assembler to produce native 68000 object code.

14.1.3 Options

-fD   Generate debugging output. The **-fD** option causes the compiler to generate the appropriate code and data for operation with the Meridian AdaVantage Debugger. For more information on using this option and using the Debugger, see Chapter 15.

-fa   Annotate assembly language listing. The **-fa** option causes the compiler to annotate an assembly language output file. The output is supplemented by comments containing the Ada source statements corresponding to the assembly language code sections written by the code generator. To use this option, the **-S** option must also be specified, otherwise the annotated file is not emitted. Refer to section 14.1.9 for more information.

-fE   Generate error log file. The **-fE** option causes the compiler to generate a log file containing all the error messages and warning messages produced during compilation. The error log file has the same name as the source file, with the extension **.err**. For example, the error log file for **simple.ada** is **simple.err**. The error log file is placed in the current working directory. In the absence of the **-fE** option, the error log information is sent to the standard output stream.

-fL   Generate exception location information. The **-fL** option causes location information (source file names and line numbers) to be maintained for internal checks. This information is useful for debugging in the event that an "exception never handled" message appears when an exception propagates out of the main program (see section 5.7). This flag causes the code to be somewhat larger. If **-fL** is not used, exceptions that propagate out of the main program will behave in the same way, but no location information will be printed with the "exception never handled" message.

-fN   Suppress numeric checking. The **-fN** flag suppresses two kinds of numeric checks for the entire compilation:

1. division_check

2. overflow_check

These checks are described in section 11.7 of the LRM. This flag reduces the size of the code. Note that there is a related ada option, −**fs** to suppress all checks for a compilation. See also section 5.7.

The −**fN** option must be used in place of pragma suppress for the two numeric checks, because presently pragma suppress is not supported for division_check and overflow_check. Pragma suppress works for other checks, as described in section 4.2. In the absence of the −**fN** flag, the numeric checks are always performed.

−**fR** Inhibit static initialization of variables. Although this option is intended for generating ROMable programs on other systems, its primary use on the Macintosh is to enforce a "code–for–data" swap, since global data space is at a premium on the Macintosh (see Chapter 11). However, the amount of global segment space saved by using this option tends to be small. String literals are always placed in the global segment, regardless of the setting of −**fR**. The −**fR** option is applicable only in the presence of the −**fs** option, which suppresses certain runtime checks. Normally, the Ada compiler initializes constants or variables with static data when the following conditions all occur:

If the −**fR** flag is specified, static initialization is suppressed for variables (but not for constants); assignments to each component of a variable are performed in the code. Note that this always happens in the absence of the −**fs** option.

−**fs** Suppress all checks. The −**fs** flag suppresses all automatic checking, including numeric checking. This flag is equivalent to using pragma suppress on all checks. This flag reduces the size of the code, and is good for producing "production quality" code or for benchmarking the compiler. Note that there is a related **ada** option, −**fN** to suppress only certain kinds of numeric checks. See also sections 4.2 and 5.7.

−**fU** Inhibit library update. The −**fU** option inhibits library updates. This is of use in conjunction with the −**S** option. Certain restrictions apply to use of this option; see section 14.1.9.

−**fv** Compile verbosely. The compiler prints the name of each subprogram, package, or generic as it is compiled.

−**fw** Suppress warning messages. With this option, the compiler does not print warning messages about ignored pragmas, exceptions that are certain to be raised at run–time, or other potential problems that the compiler is otherwise forbidden to deem as errors by the LRM.

−**g** The −**g** option instructs the compiler to run an additional optimization pass. The optimizer removes common sub–expressions, dead code and unnecessary jumps. It also does loop optimizations. This option is different from the −**g** option to **bamp**. The −**g** option to **ada** optimizes the specified unit when it is compiled; no inter–unit optimization is done. The −**g** option to **bamp** analyzes and optimizes the entire program at link time. Note that even if −**g** is specified for the **ada** command, the −**K** option to **ada** must still be specified for the −**g** option to **bamp** to be effective.

−**help**

The −**help** option is used to invoke the Meridian AdaVantage help facility. You must have Apple's HyperCard application installed. The −**help** option must be the first option on the **ada** command line.

−**K** Keep internal form file. This option is used in conjunction with the Optimizer (see Chapter 9 for more information). Without this option, the compiler deletes internal form files following code generation.

−**l**modifiers

Generate listing file. The −**l** option causes the compiler to create a listing. Optional modifiers

may be given to affect the listing format. You can use none or any combination of the following modifiers:

- **c**   continuous listing format.
- **p**   obey pragma **page** directives.
- **s**   use standard output.
- **t**   relevant text output only.

The formats of and options for listings are discussed in section 14.1.7. The default listing file generated has the same name as the source file, with the extension **.lst**. For example, the default listing file produced for **simple.ada** has the name **simple.lst**. The listing file is placed in the current working directory. Note: **-l** also causes an error log file to be produced, as with the **-fE** option.

-L *library-name*

Default: **ada.lib**

Use alternate library. The **-L** option specifies an alternative name for the program library.

-mbg   The **-mbg** option prepares a compilation unit for use with the MacsBug machine-level debugger. Symbols are truncated to 31 characters. Refer to the *Macintosh Programmer's Workshop Reference* for information about MacsBug.

-mc68020*

The **-mc68020** option causes MC68020 processor instructions to be emitted. Warning: This renders a program inoperable on any Macintosh systems but those equipped with the 68020 or 68030 processor.

-N   No compile. This option causes the **ada** command to do a "dry run" of the compilation process. The command invoked for each processing step is printed. This is similar to the **-P** option, but no actual processing is performed.

-P   Print compile. This option causes the **ada** command to print out the command invoked for each processing step as it is performed.

-S   Generate assembly code. Causes the code generator to produce an assembly language source file and to halt further processing. This option must be used cautiously; refer to section 14.1.9 to find out how to use this option properly.

Note: Options beginning with **-f** may be combined, as in "**-fsv**." This is equivalent to specifying the options separately, e.g. "**-fs -fv** ." Options beginning with **-l** may be similarly combined or separated, as in "**-lcs**" or "**-lc -ls**" (see section 14.1.7).

14.1.4   Compiler Output Files

Files produced by compilations, other than listings and error logs, are:

.a files   assembly language files (only when **-S** option used)

.a.o files   object code files

.atr files   interface description files

---

*This option is actually processed by the Assembler. Although all the MPW   Assembler options are accepted and passed along to the Assembler by the **ada** command, only the Assembler options useful in conjunction with Meridian AdaVantage are discussed. For additional details of Assembler options, refer to the *Macintosh Programmer's Workshop Reference*. Unless otherwise noted, "the compiler" or "the code generator" refer to components of the AdaVantage compiler, not to the Assembler.